PATENT
Attorney Docket No.: 100111238-1
Express Mail Label No.: EV 008783480 US

# SYSTEM AND METHOD FOR ITERATIVELY TRAVERSING A HIERARCHICAL CIRCUIT DESIGN

## RELATED APPLICATIONS

**[0001]** The present document contains material related to the material of copending, cofiled, U.S. patent applications Attorney Docket Number 100111221-1, entitled System And Method For Determining Wire Capacitance For A VLSI Circuit; Attorney Docket Number 100111227-1, entitled System And Method For Determining Applicable Configuration Information For Use In Analysis Of A Computer Aided Design; Attorney Docket Number 100111228-1, entitled Systems And Methods Utilizing Fast Analysis Information During Detailed Analysis Of A Circuit Design; Attorney Docket Number 100111230-1, entitled Systems And Methods For Determining Activity Factors Of A Circuit Design; Attorney Docket Number 100111232-1, entitled System And Method For Determining A Highest Level Signal Name In A Hierarchical VLSI Design; Attorney Docket Number 100111233-1, entitled System And Method For Determining Connectivity Of Nets In A Hierarchical Circuit Design; Attorney Docket Number 100111234-1, entitled System And Method Analyzing Design Elements In Computer Aided Design Tools; Attorney Docket Number 100111235-1, entitled System And Method For Determining Unmatched Design Elements In A Computer-Automated Design; Attorney Docket Number 100111236-1, entitled Computer Aided Design Systems And Methods With Reduced Memory Utilization; Attorney Docket Number 100111257-1, entitled Systems And Methods For Establishing Data Model Consistency Of Computer Aided Design Tools; Attorney Docket Number 100111259-1, entitled Systems And Methods For Identifying Data Sources Associated With A Circuit Design; and Attorney Docket Number 100111260-1, entitled Systems And Methods For Performing Circuit Analysis On A Circuit Design, the disclosures of which are hereby incorporated herein by reference.

## BACKGROUND

**[0002]** An electronic computer aided design ("E-CAD") tool is used to create and analyze a circuit design, including a very large scale integration ("VLSI") circuit design. The circuit design includes a netlist that identifies electronic design elements (e.g., capacitors, transistors, resistors, etc.) and their interconnectivity (e.g., nets) within the circuit design.

**[0003]** A signal net is a single electrical path in a circuit design that has the same electrical characteristics at all of its points. Any collection of wires that carries the same signal between design elements is a signal net. If the design elements allow the signal to pass through unaltered (as in the case of a terminal), then the signal net continues on subsequently connected wires. If, however, the design element modifies the signal (as in the case of a transistor or logic gate), then the signal net terminates at that design element and a signal new net begins on the other side.

**[0004]** A significant characteristic of VLSI and other types of circuit design is a reliance on hierarchical description. A primary reason for using hierarchical description is to hide the vast amount of detail in a circuit design. By reducing the distracting detail to a single object that is lower in the hierarchy, one can greatly simplify many E-CAD operations. For example, simulation, verification, design-rule checking, and layout constraints can all benefit from hierarchical representation, which makes them more computationally tractable. Since many circuit designs are too complicated to be easily considered in their totality, a complete design is often viewed as a collection of design element aggregates that are further divided into sub-aggregates in a recursive and hierarchical manner. In VLSI circuit design, these aggregates are commonly referred to as design blocks (or cells). The use of a design block at a given level of hierarchy is called an 'instance'. Each design block has one or more 'ports', each of which provides a connection point between a signal net within the design block and a signal net external to the design block. A net within one design block may thereby connect with a net in another design block, the net 'pieces' forming a single net known as a 'highest level signal name' ("HLSN"). The HLSN is identified by the name of the net 'piece' located at the highest hierarchical level in the circuit design.

[0005] When tracing signal paths through a hierarchical circuit design, it is often desirable to know the HLSN of a given net, other than the one being traced. This situation can occur, for example, when a terminal of a device is reached at some level of the hierarchical design, and the HLSN of one or more other terminals for that device needs to be determined. Presently known tracing strategies typically use recursive methods that employ complex algorithms that redundantly analyze a circuit design, and which are conceptually non-intuitive to developers of E-CAD tools.

[0006] Tracing a hierarchical circuit design can be a tedious process to implement. The following pseudo-code illustrates one manner in which the trace has typically heretofore been performed. Initially, an initial net is selected. Then, a routine (such as 'RECURSE', below) is employed to recursively traverse the hierarchical design for each port and port instance on the net.

```
RECURSE:
DoSomeStuff(net)

foreach portinst on net {
    get instance owner of portinst
    get describer of instance
    get port describer of portinst
    push instance onto instance_hist
    get net on port
    RECURSE(net, instance_hist)
}

foreach port on net {
    get instance from instance_hist
    get portinst instantiation of port
    pop instance off instance_hist
    get net on portinst
    RECURSE(net, instance_hist)
}
END RECURSE
```

[0007] The above recursive routine is executionally redundant, and therefore not particularly efficient, and can thus require a great deal of processor time

to traverse a typical VLSI circuit design, particularly when an analysis tool needs to iterate over nets in the design to determine properties or quantities unrelated to the circuit hierarchy. The problem of traversals can become even more complicated when the capability of skipping 'around' elements in the design is desired. For instance, if a field-effect transistor ("FET") is encountered in tracing through a circuit design, it may be necessary to skip to the opposite side of the FET (e.g., source to drain, or drain to source) and continue tracing.

## SUMMARY

**[0008]** In one embodiment, a method iteratively traverses a hierarchical circuit design. An initial net, and an instance history that uniquely defines the initial net within the circuit design are selected. The initial net and the instance history are appended to a list of nets to be processed. The initial net and the instance history are inserted into a set of visited nets. Each additional net connected to the initial net is visited in response to a first request from a user. The initial net and each additional net are returned in response to a second request from the user.

**[0009]** In another embodiment, a system iteratively traverses a hierarchical circuit design, including: means for selecting an initial net and an instance history that uniquely defines the initial net within the design; means for appending, to a list of nets to be processed, the initial net and the instance history; means for inserting, into a set of visited nets, the initial net and the instance history; and means for visiting, in response to a first request from a user, each additional net connected to the initial net.

**[0010]** In another embodiment, a system iteratively traverses a hierarchical circuit design. The system includes an iterator function, an incomplete trace object, and a processor. The iterator function selects an initial net and an instance history that uniquely defines the initial net within the circuit design. The iterator function appends the initial net and the instance history to a list of nets to be processed. The iterator function inserts the initial net and the instance history into a set of visited nets. The incomplete trace object visits each additional net connected to the initial net in response to a first request from a user. The incomplete trace object returns the initial net and each additional net in response to a second request from the user. The processor executes the iterator function and invokes the incomplete trace object.

[0011] In another embodiment, a software product comprises of instructions, stored on computer-readable media, wherein the instructions, when executed by a computer, iteratively traverses a hierarchical circuit design, comprising: instructions for selecting an initial net and an instance history that uniquely defines the initial net within the design; instructions for appending, to a list of nets to be processed, the initial net and the instance history; instructions for inserting, into a set of visited nets, the initial net and the instance history; instructions for visiting, on a first request from a user, each additional net connected to the initial net; and instructions for returning, on a second request from the user, the initial net and each additional net.

## BRIEF DESCRIPTION OF THE DRAWINGS

[0012] Figure 1 shows one system for iteratively traversing a hierarchical circuit design.

[0013] Figure 2 is a flowchart illustrating an exemplary set of steps performed during operation of the system of Figure 1.

[0014] Figure 3 is a flowchart illustrating an exemplary set of steps performed to facilitate the operation shown Figure 2.

[0015] Figure 4 is a diagram of an exemplary hierarchical design that may be traversed using the present method.

[0016] Figure 5 is a flowchart illustrating one process for iteratively traversing a hierarchical circuit design.

## DETAILED DESCRIPTION

[0017] Figure1 shows an exemplary electronic computer aided design ("E-CAD") system 100 configured for iteratively traversing a hierarchical circuit design 109. In one example, system 100 traverses design elements and/or nets of circuit design 109. System 100 includes computer system 101 and E-CAD tool 107. Computer system 101 controls E-CAD tool 107 to analyze circuit design 109, which includes a hierarchical VLSI design description, typically including a netlist 120 defining design elements and nets associated with circuit design 109. Computer system 101 includes processor 102, computer memory 104, and storage unit 106. Processor 102 is coupled to computer memory 104 and to storage unit 106. In one

embodiment, E-CAD tool 107 initially resides in storage unit 106. Upon initialization, E-CAD tool 107 and at least a part of design 109 are loaded into computer memory 104. Processor 102 then executes E-CAD tool 107 to perform the functions described herein.

**[0018]** Circuit design 109 may be viewed as a collection of components that are further divided in a hierarchical manner into sub-components commonly referred to as blocks or cells. The use of a block at a given level of hierarchy is called an 'instance'. Each block in design 109 has one or more 'ports', each of which provides a connection point between a net within the block and a net external to the block. The terms 'block', 'instance', 'port', and 'port instance' may be explained as follows, using exemplary circuit design 400 in Figure 4 as a reference. As shown in Figure 4, circuit design 400 includes block instances i0, i1, and i2. Instances i0 and i1 are hierarchically connected by nets 'a1', 'a2', and GND. Instances i1 and i2 are hierarchically connected by nets 'pass', 'up_vdd', and 'dn_gnd'. Instance i1, when viewed from the perspective of a top level in the sub-hierarchy represented by test_block_i1 and test_block_i2, is itself considered to be a block, i.e., test_block_i1, in that particular hierarchical context. In the same context, the block test_block_i1 contains instance i2, which, if considered from a perspective internal to the 'box' (i2 / test_block_i2) shown in Figure 4, would be referred to as block test_block_i2.

**[0019]** It can thus be seen that the definitions of 'block' and block 'instance' respectively depend on whether a particular 'box' (a block or instance of the block) is viewed from an internal or external standpoint, i.e., the appropriate nomenclature depends on the hierarchical perspective from which the 'box' is viewed. Each 'box' (block or instance) has a plurality of 'ports' and corresponding port instances ('portinsts'), each pair of which provides a connection point between a net within the block and a net external to the block. A 'port/portinst' thus comprises two contiguous parts, a first part, termed a 'portinst', which is a port instance located externally on a box (or instance) boundary; and a second part, termed a 'port', which is located internally on the box (or block) boundary.

**[0020]** As can be seen from Figure 4, portinsts are the half of the 'port/portinst' on the outside of a 'box' (for example, item 412), and ports are the half of the 'port/portinst' on the inside of a 'box' (e.g., item 413). A port takes the same

name as the net to which it is connected, and port instances have the same name as their describing port (i.e., the same name as the net in the describing block). When examining netlist 120 that is part of design 109, portinst 412 in design 400 may be described in netlist 120 as 'net pass –> port inst in' in the block test_block_i1. This netlist entry indicates that the net 'pass' connects to the portinst 'in' 412 on instance 'i2.' The corresponding port in test_block_i2 is port 413, which has the name 'in', since it is connected to net 'in' in test_block_i2. In an exemplary embodiment, a hierarchical model stored as part of design 109 is used to represent the hierarchy of design 400, and the difference between a portinst and a port is readily determined through the use of object-oriented techniques in which port and portinst are different objects that are owned by different types of objects.

**[0021]** With further regard to figure 1, system 100 provides a mechanism for iteratively traversing a path through nets in a hierarchical design, without redundantly returning or identifying nets encountered in the trace process. System 100 also provides the capability of skipping 'around' instances in a hierarchical design 109 when tracing through circuit design 109. A node is a connectivity point within the netlist (e.g., the FET source is connected to a first node, and the FET drain is connected to a second node). For example, if a FET is encountered in tracing through design 109, it may be desirable or necessary to skip to the opposite side (node) of the FET (source to drain, or drain to source) and continue tracing. The present system 100 may therefore include an 'identity' function 111 that causes system 100 to traverse around blocks indicated by the user, to avoid entering certain blocks. This allows tracing across transistor channel connections, for example, which allows specifying the specific portinsts to be traced out of. A hierarchical net iterator ('HierNetIter') function 105 may invoke additional functions 'IdentFunc' and 'HandleFunc', as described below:

```
HierNetIter(net, instance_hist, IdentFunc, HandleFunc)
```

**[0022]** In one example, as E-CAD tool 107 initially traverses a signal from net to net through associated ports and port instances, a pointer to an object that represents the currently encountered instance is pushed onto a stack or list (an 'instance history list') 108, containing instances of blocks (hereinafter simply

'instances') that have been traced 'into'. Instance history list 108 is then used to retrace the path back up to higher levels of the hierarchy. The instance history in list 108 ('instance_hist') is used to uniquely identify the starting net that was reached in the trace process. The present system 100 may also utilize user-defined 'callback' functions to handle special cases encountered during traversal of circuit design 109. For example, identity function ('IdentFunc') 111 may be used to provide an indication of which instances to skip across (trace around), and a handle function ('HandleFunc') 112 may be used to identify which portinsts (with associated nets) on identified instances should be added to the list of connected nets (e.g., non-driver or receiver FETs may be crossed to the other channel connected node, and tracing may be stopped at any driver or receiver terminal).

**[0023]** Functions 105, 111, 112 may provide certain flexibility in traversal of circuit design 109. For example, assume that E-CAD tool 107 starts at a driver output (e.g., net 'out2' in instance i2 in the example shown in Figure 4), and traces to all connected nets while crossing any channel connected FETs (e.g., FET1 and FET2). Assume, also, that this traversal should terminate when encountering the gate connection of each FET, and not trace arou ıd it. Hierarchical net iterator function 105 allows this type of circuit traversal to be performed, for example in the following manner:

ITERATOR ROUTINE EXAMPLE

```
HierNetIter hni(out2, instance_hist_to_out2,
IsTransistor,

StopAtDriverOrReceiver);
for (hni.Begin(); !hni.End(); hni++) {
    local_net = *hni;
    PerformOperation(local_net);
}
```

**[0024]** As the 'for' loop shown above in the Iterator Routine Example is iterated, net iterator function 105 visits each unique hierarchical net in the circuit design to which it is applied, starting with net 'out2' in this particular example. Since a hierarchical net is unique relative to a specific selected net, as well as its associated

instance history in a given trace, the iterator function 105 returns both of these entities.

## HIERARCHICAL NET ITERATOR FUNCTION 105

**[0025]** In one embodiment, the hierarchical net iterator functionality utilizes 'incomplete traces', which are represented by objects ('incomplete trace objects' 110) that represent a particular (unique) net in a hierarchical design, and which maintain information about what other connections on this net need to be traced. In an exemplary embodiment, hierarchical net iterator 105 performs the operations shown in the flowchart of Figure 2, in the following manner:

**[0026]** At step 201, prior to initialization of net iterator function 105, a signal is traced from a starting net in design 109 to a terminal net in the design. As this initial trace is performed, the signal is traced from net to net through associated ports and port instances, and a pointer to an object that represents the instance of interest is pushed onto a stack or list ('instance history list') 108, containing instances that have been traced 'into'. Instance history list 108 is then used to retrace the path back up to higher levels of the design hierarchy.

**[0027]** At step 205, net iterator function 105 is initialized with an initial net and an instance history that uniquely defines the path traversed through design 109 to reach the initial net. In the example 'HierNetIter' function call shown below, the parameters provided to net iterator function 105 are:

'instance_hist_to_out2' – the trace history of the initial trace up to net 'out2', stored in instance history list 108;

'IsTransistor' – the identity function 111 parameter indicating that each instance of a transistor (e.g., a PFET or NFET) is to be traced around; and

'StopAtDriverOrReceiver' – the handle function 112 parameter indicating that the trace should cross to the other node for channel connected non-driver or receiver transistors, and stop tracing at any driver or receiver terminal.

```
HierNetIter hni(out2, instance_hist_to_out2, IsTransistor,
                StopAtDriverOrReceiver)
```

**[0028]** In step 210, net iterator function 105 then creates an incomplete trace object 110 with the specified net and instance history, and initializes the portinst

and port iterators (described with respect to Figure 3, below) within the object. This first incomplete trace object 110 is placed on top of an 'incomplete trace' stack 115 in computer memory 104, at step 215. At step 217, the incomplete trace object 110 thus created is inserted in a set of visited incomplete trace objects 113.

**[0029]** At step 220, each time a user requests the net that net iterator function 105 is presently 'pointing' to, iterator function 105 is dereferenced (step 225), and returns a structure containing the net and instance history that are within the incomplete trace object 110 currently on the top of the incomplete trace stack 115, at step 230.

**[0030]** At step 240, each time a user requests the next net relative to the current incomplete trace object 110, iterator function 105 is incremented, at step 245, and the next incomplete trace object 110 on the top of the stack 115 is requested, at step 250, to provide the next unique net that can be reached from the net within the incomplete trace object 110. Thus, when iterator function 105 is incremented, it receives the next connected net from incomplete trace object 110, given its current state.

**[0031]** At step 255, if the top incomplete trace object 110 returns unsuccessfully, i.e., if the incomplete trace object has reached the end of all of its lists (other port insts, port insts, and ports, as indicated in Figure 3), there is nothing more that can be traced within that incomplete trace object, so it is now 'complete' and can be removed from stack 115. Net iterator function 105 removes the top incomplete trace from the incomplete trace stack 115 and deletes it, at step 260. Processing continues with step 265, described below.

**[0032]** If the incomplete trace object returns successfully (at step 255), then at step 257, if the returned incomplete trace has not been visited, i.e., if the current incomplete trace object 110 is not in the set of visited incomplete trace objects 113, it is added to the incomplete trace stack 115, at step 215, at which processing continues; otherwise, the incomplete trace object 110 returns the next unique net in the incomplete trace object, at step 250, described above in connection with figure 3.

**[0033]** If, at step 265, the incomplete trace stack 115 is empty, then the net iterator function's task is completed (step 270), and successive calls to increment or dereference iterator function 105 will not change its state. If the incomplete trace

stack 115 is not empty, then subsequent 'increments' to Net iterator function 105 enter at the increment entry point 245, and 'dereferences' enter at the dereference entry point 225.

GET_NEXT_NET FUNCTION

**[0034]** The creation of an incomplete trace object 110 involves initializing 'sub-iterators' that check all of the portinsts and ports connected to the given net. The portinst and port iterators (described below) iterate over a collection of items (portinsts or ports). Figure 3 is one embodiment of block 250 in Figure 2, in which an incomplete trace object 110 uses an algorithm to determine the next net in the trace process. This algorithm (the 'Get_Next Net' function) is described as follows:

**[0035]** In response to a user request (step 300) to find the next net in a trace operation, a check is made at step 305 to determine whether there are portinsts in the other_port_insts list 114 (see step 340, below). If list 114 is not empty, then at step 310, the next portinst is removed from list 114, and, at step 315, the net connected to the portinst, along with the current instance history list 108, is returned, since the instance history uniquely identifies the current net.

**[0036]** At step 320, if the port inst iterator (represented by blocks 305–340) is at the end of its collection, processing continues with step 350, below. If the port inst iterator is not yet at the end of its collection, then iterator function 105 is incremented at step 325, and the port instance, to which iterator function 105 is pointing, is retrieved.

**[0037]** At step 330, handle function 112 is invoked to check the instance that owns the portinst (the 'owning instance') to see if it is a 'special case'. Special cases are determined and processed by a user-provided callback function to allow the net iterator 105 to handle certain types of instances in a manner defined by the user. For example, a particular user might require a function that skips from one side of a FET to the other. Identity function ('IdentFunc') 111 is an example of a function that handles such a 'special case'. If the present owning instance is a special case, then, at step 335, the list of portinsts to be handled (using handle function 112) is retrieved. The list is then assigned to the other_port_insts list 114 (Figure 1), at step 340, and the above-described part of the algorithm is repeated, starting with step 305.

**[0038]** If, at step 330, the owning instance is not a special case, then at step 332, the port that describes the current portinst is retrieved. At step 333, the instance that owns the port instance is pushed onto a stack containing a copy of instance history list 108; and, at step 334, the net connected to the port, along with this copy of instance history list, is returned.

**[0039]** At step 350 (when the port inst iterator is at the end of its collection), if the port iterator (represented by blocks 350–380) is not at the end of its collection, then, at step 365, the port iterator is incremented, and the port to which the iterator is pointing is retrieved. At step 370, a copy of instance history list 108 is made, and the top instance is popped off of the copy. At step 375, the portinst on the instance that is an instantiation of the port is retrieved, and the net connected to that portinst, along with the copy of the instance history list 108, is returned at step 380.

**[0040]** At step 350, if the port iterator is at the end of its collection, the other_port_insts list 114 is empty, and thus both the portinst iterator and the port iterator are at the end of their respective collections. Since no further nets are to be found, this 'incomplete' trace is now complete. Thus, at step 360, a 'failure to select another net' indication is returned.

**[0041]** The present net iterator function 105 allows use of other desired design models by using a new definition of incomplete trace object 110. For example, system 100 may employ a model wherein nets are not the primary connection method, but instead has multiple owned connection devices.

**[0042]** It should be noted that either the incomplete trace object or the iterator function may perform the function of retrieving the next net, i.e., the function of performing next-net-retrieval is not limited to inclusion in the incomplete trace object, and this function may be performed in the iterator or elsewhere in a separate function.

**[0043]** Figure 5 is a flowchart illustrating one process 500 for hierarchically traversing a circuit design. Process 500 is for example implemented by system 100, figure 1. In step 502, process 500 selects an initial net and an instance history that uniquely defines the initial net within the circuit design. In step 504, the initial net and the instance history are appended to a list of nets to be processed. In step 506, the initial net and the instance history are inserted into a set of visited nets.

In step 508, each additional net connected to the initial net is visited in response to a first request from a user,. In step 510, the initial net and each additional net are returned in response to a second request from the user.

**[0044]** Instructions that perform the operation discussed with respect to Figures 2, 3 and 5 may be stored on computer-readable storage media. These instructions may be retrieved and executed by a processor, such as processor 102 of Figure 1, to direct the processor to operate in accordance with the present system. The instructions may also be stored in firmware. Examples of storage media include memory devices, tapes, disks, integrated circuits, and servers.

**[0045]** Certain changes may be made in the above methods and systems without departing from the scope of the present system. It is to be noted that all matter contained in the above description or shown in the accompanying drawings is to be interpreted as illustrative and not in a limiting sense. For example, the items shown in Figure 1 may be constructed, connected, arranged, and/or combined in other configurations, and the set of steps illustrated in Figures 2 and 3 may be performed in a different order than shown without departing from the spirit hereof.